|  |
| --- |
| Circle Language Spec: Commands |

## Commands Main Concepts

Commands may be objects, that so happen to be *executable*. Commands could be actions, procedures and processes, that a computer might perform. They might be displayed in a diagram as squares and diamond shapes:

|  |  |
| --- | --- |
|  |  |

### Diagram Elements

The concept of commands may boil down to a limited set of characteristics.

A *square* in Circle might symbolize that the command is **not** **executable**.



A *diamond* in Circle may express a command that could be**executable**.



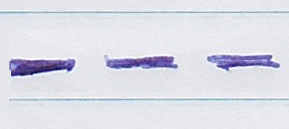
Containment is a way to express a relationship. One symbol might **contain** another:



A *solid line* between symbols might mean that one command is a **reference** to another:

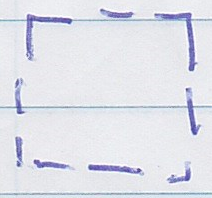


A *dashed line* between command symbols might make one command a kind of copy of another:



It might also be said, that it indicates one command's being the **prototype** for another.

If a command would *only* be used as a prototype, it might be drawn with a *dashed border*:



(Using dashed shapes is still a bit of an open discussion.)

A command may have a *name*:



It might also be *nameless.*



Here is an attempt to summarize these traits that commands might have:

* **(not) executable**
* **(not) named**
* **containment**
* **references**
* **prototypes** (or "definitions")

### Derived Constructs

Constructs that might be known from other programming languages, may have a reasonably unique expression using combinations of these more basic elements from Circle mentioned above. Here follows an attempt to accompany this claim with some examples.

### Command Definitions

A command definition might describe the structure and behavior of another command.

It may look as follows in another programming language:

**void MyDefinition()**

**{**

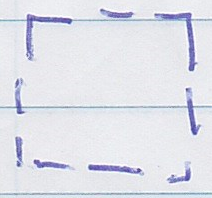
**}**

To express it in Circle, a square might be used:



Its being a square, might indicate that it may *not execute*.

When *only* used as a definition, it might be drawn out with a *dashed border* too:

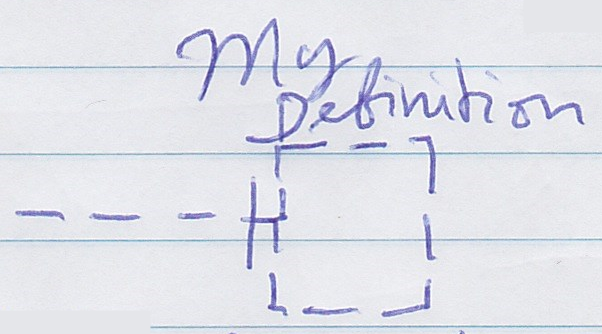


(Though dashed shape notation is still a bit of an open discussion.)

A definition might also have a *name*:



When used, a command definition may be pointed to by a *dashed line*:



That way another symbol might use **MyDefinition** as a *prototype*.

So a command definition might be symbolized by:

* square
* named
* dashed
* pointed to with dashed lines

because it may be:

* not executable (directly)
* named
* used as a prototype

### Command Calls

A command definition might be *called* multiple times.

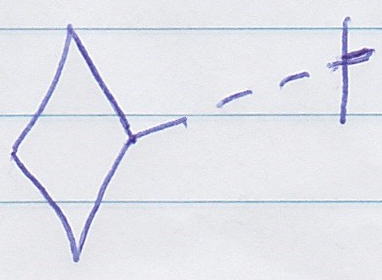
In a different programming language a command call might look as follows:

**MyCommand();**

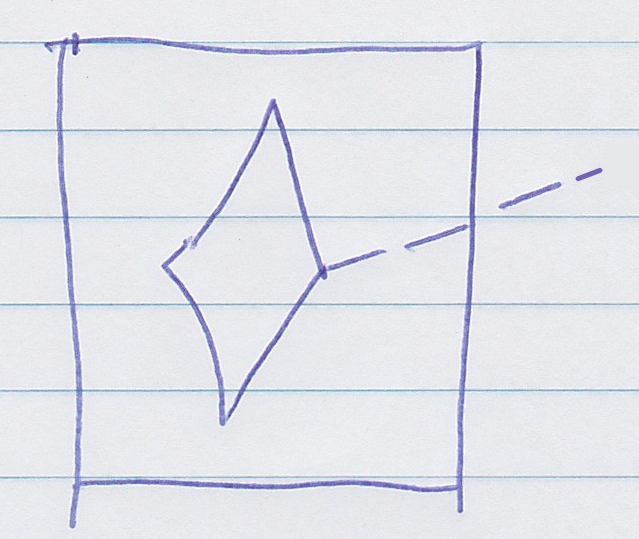
A command call may *execute*, so it might be symbolized with a diamond shape:

|  |
| --- |
|  |

A call may point out a definition, so it might have a *dashed line* pointing away from it:



A call might be placed *inside* a parent command.



The call itself might remain *nameless*.

So a command call might be symbolized by:

* diamond
* nameless
* contained in a square
* dashed line pointing away from it

because it may be:

* executable
* nameless
* contained inside a definition
* uses another command as a definition

### Command References

A command could be *pointed to*.

This may look as follows in another programming language:

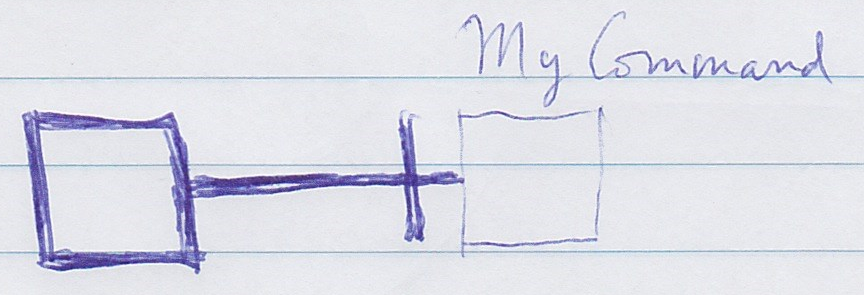
**MyCommand**

So without any decoration with brackets or what have you, it may represent a reference to a command.

A command reference may be a square, to indicate it does not *execute* directly.



A *solid line* can be used to point out a different command:



So a command reference might be symbolized by:

* square
* solid line pointing to another command

because it is:

* not executable
* a reference

### Code Blocks

Some programming languages might allow code blocks within a command, to group statements together. This might scope variables, so variables inside the block might only be used within that block, for some added safety against programming errors, perhaps.

In another programming language, this might look as follows:

void MyDefinition()

{

**{**

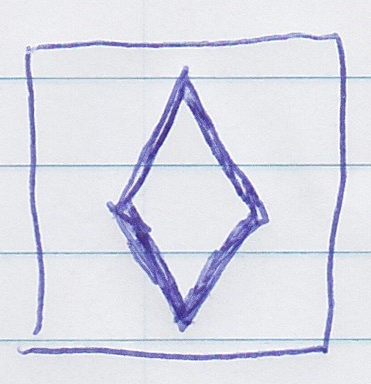
**…**

**}**

}

The inner braces and its content would be the code block.

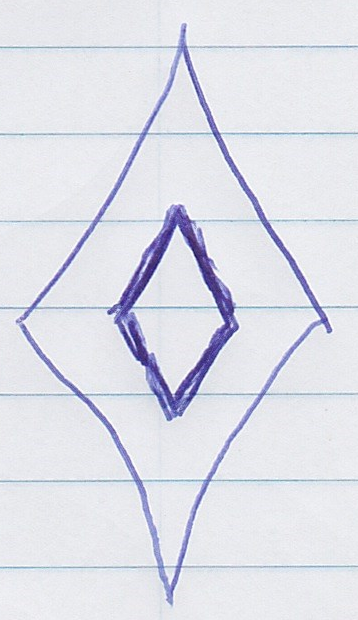
In Circle, a code block may be *embedded* inside another command:



The inner command (the code block) might be drawn as a *diamond*: it *executes* if the parent command executes.

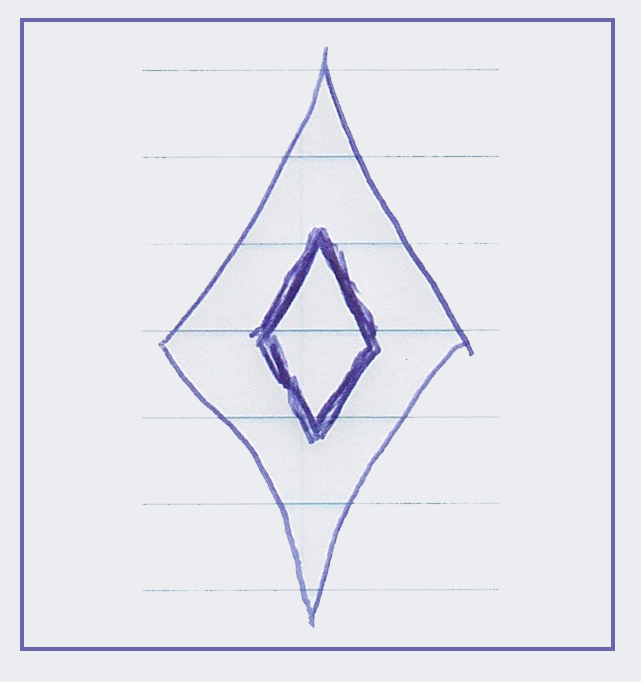
A code block might be *nameless*. It might not have any *lines* going towards it or away from it.

A code block might also look like this:



The previous diagram might show a code block within a definition (a square). This diagram shows the code block while that command is executed.

Code blocks might be nested even further:



So a code block might be symbolized by:

* a diamond
* inside another command
* nameless
* no lines pointing to or from it

because it is:

* executable
* contained by another command
* nameless
* not a reference
* not a definition

### Local Function

A local function might be known from other programming languages and may mean a command that is defined within another command.

In a different programming language this might look as follows:

void MyParentCommand()

{

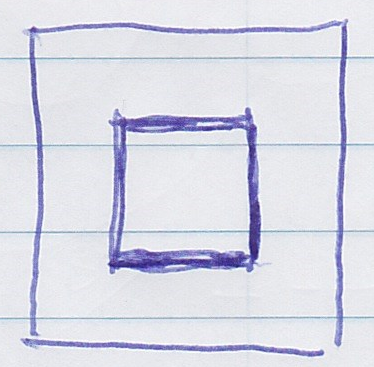
**void MyLocalFunction()**

**{**

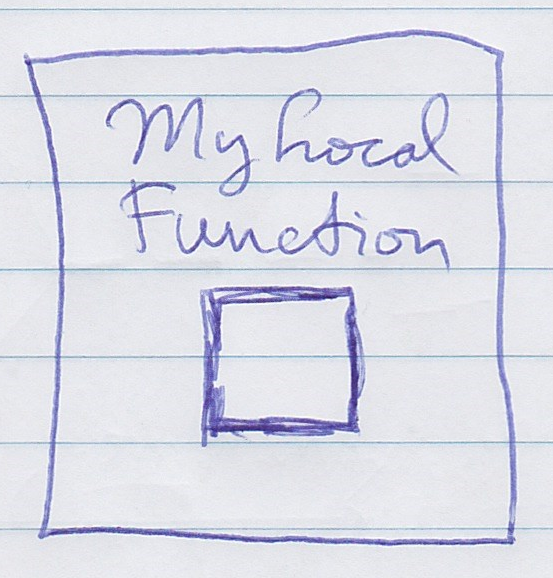
**}**

}

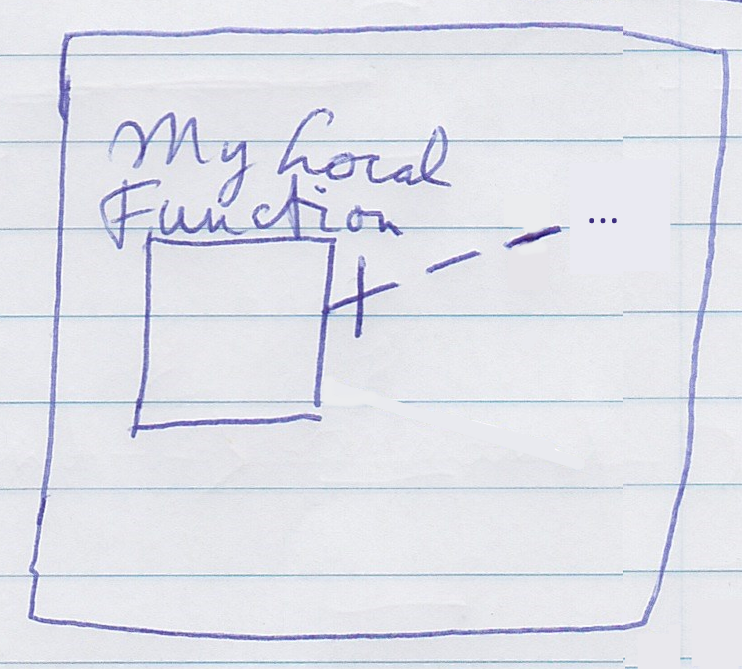
In Circle it might also be a command definition contained within another command definition.



When peeking at another programming language, a local function seems to be able to have a name.



It may also be *called* from within its parent command:



So a local function might be symbolized by:

* a square
* within a square
* with a name
* pointed to by dashed lines (perhaps)

because it is:

* not executed directly
* contained in a definition
* has a name
* a prototype / might be called

### Clauses

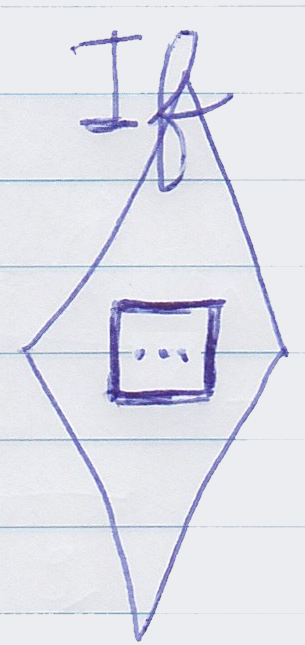
Clauses might be found as elements of an **if** statement or for instance a **where** clause from querying languages.

This might look as follows in another language:

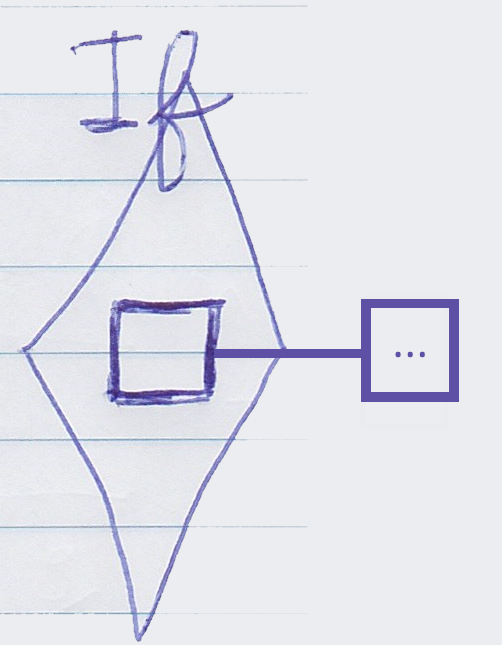
if ***{ … }***

(This is pseudo code. A full **if** statement might have more parts to it.)

There might be a bit of an interpretation problem here. A clause in an **if** statement looks a bit like a *code block*. Maybe it could be interpreted as such.



It also may look a bit like a nameless local function that you might pass a reference to, to the **if** statement:



The following was another attempt to define what a clause might be in Circle: *a command inside another command with no lines going to or from it*. That definition might not cover it. It might be just terminology quibbles. Later on it seemed to not matter much. Command-like constructs from other languages seem to be expressible within Circle perhaps in a good enough fashion.

So a clause might be expressed with:

* a square
* inside a diamond
* that might connect to another square
* with a solid line

because it is

* not executable (not directly)
* passed to another command or statement
* might be a command reference

### Lambda Expression

A lambda expression might be found in other programming languages as sort of a short notation for a function, embedded inside other statements.

In another programming language this might look as follows:

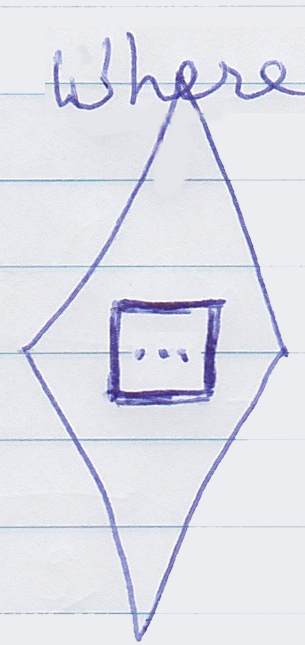
Where(**x => x.IsChecked**)

Minus the details:

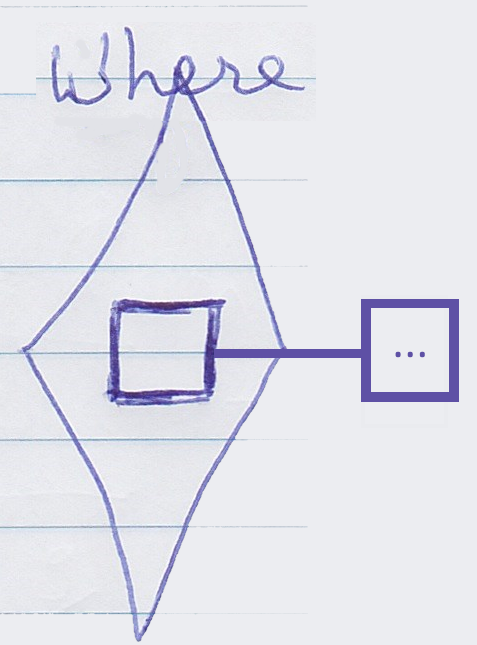
Where(**…**)

So the … would be the lambda expression.

Honestly said, in Circle there might not be a way to distinguish them from *clauses*.



Or:

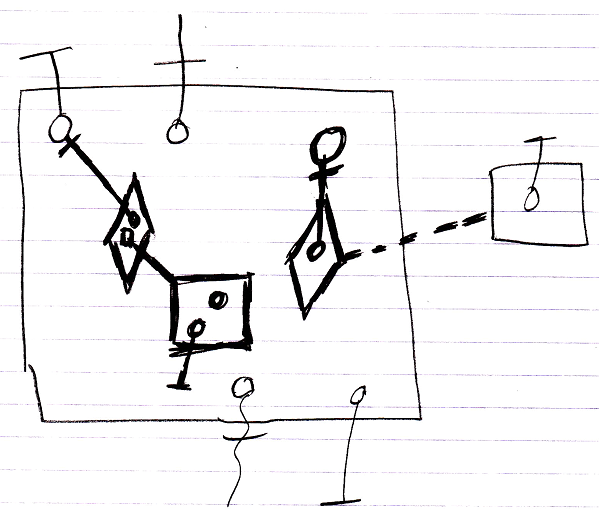


### Parameters and Return Values Not Covered

Symbolizations for parameters and return values might be missing in much of these texts. Those are intended to be covered by another chapter: "Parameters".

### Implementation

The implementation of a command might be defined as the private contents of a command. Here is an attempt to demonstrate that visually:



The contents of the large square, drawn with thick lines, would represent the command’s *implementation*, because they would be private. The objects inside the large square that are drawn with thinner lines, might be the command’s parameters: its publics. (Notation for *parameters* might be involved in the picture above, but might remain unexplained until the chapter "Parameters".)

### Conclusion

Using the constructs for commands from Circle, it may seem circumstantial whether a command symbol could be stereotyped as a block, clause, definition, call, etc. It seems to result from the way basic language elements from Circle relate to each other. Still, it may be relevant to be aware how these symbols from Circle can be combined to represent constructs from other languages.

### Details

#### Command Definition Synonyms

* prototype
* blue-print
* inactive command
* inactive command symbol
* definition
* command definition

#### Definition Line Synonyms

* command definition line
* call line
* prototype line
* dashed line
* definition line

#### Executable Command Synonyms

* active command
* active command symbol
* executable command
* executable command symbol

#### Command Call Synonyms

* execution

#### Command Reference Synonyms

* function pointer
* method reference
* delegate

## Loose Ideas

### Command Definition

A command definition might be a blue-print for another command. A command might select another command to function as its *prototype*. The command definition may describe the procedure of a command and

<< nice formulation >>

Command definitions themselves might not necessarily be executed. Just copies of it, more likely. A command definition's not being executable might be expressed by using a square, rather than a diamond:



<< nice formulation >>

If a command is only used or usable as a definition, it might be drawn with a dashed line:

<Picture?>

<< detail >>

That might mean that the dashed line may say it is *used as a definition*, while its being a square may mean that it is *not executed*. So two characteristics symbolized with two different things.

<< already covered >>

If a command definition is used, it might be indicated with a dashed line:

<Picture?>

<< detail >>

The square on the right might server as the definition for the command on the left.

### Command Definition Edge Cases

<< detail >>

In Circle, any command might server as a command definition, so both *squares and diamonds* might be ok:

|  |  |
| --- | --- |
|  |  |

### Command Definition Compared to Objects

<< commands compared to objects >>

A command object might have a similar structure as its definition, but not necessarily the same data. Values might change for each individual command object. *Which* objects are referenced might also be different for each individual command object. But initially the command object might be an exact replica of the definition. The definition’s attribute values and object references might only function as a default.

### Executable Commands

<< nice formulation >>

An *executable* command might be carried out, while an *inactive* command, might stay asleep.

<< synonym >>

An executable command might also be called an *active command*. In

<< already covered >>

a diagram an executable command symbol might be displayed as a diamond shape:



A diamond shape could be the symbol for *execution*.

<< synonym >>

A diamond shape might also be called an *active command symbol*.

### Inactive Command

<< nice formulation >>

An inactive command object may be asleep and might never be executed.

<< already covered>>

It could be used as a prototype for another command.

<< already covered>>

In a diagram an inactive command could be displayed as a square:



<< synonym >>

A square might be a symbol for a command's being *inactive*.

### Command Call Compared to Objects

<< commands compared to objects >>

A command call might be like an instantiation of a command definition. A command call might be an individual object with something similar to a class reference to another command. There might be an analogy between calls to a definition and objects of a class, with an addition, that a call might be *executable*.

A command call might select its command definition with a *class* redirection, because the definition might be like the command call’s *prototype*, and the call may always be its own individual object for which the command definition may be the prototype.

### Command Call Behavior

<< commands compared to objects >>

Initially, a call might be sort of a copy of its definition. Data of the command definition might just be default values. Data of a call object might be changed before it might run and change while it runs. What data of a command might be changed or not, might be covered later.

<< creation behavior of commands >>

At first a command call might be sort of asleep. That might be when there is a chance to set its parameters. After that the command call might be run.

<< detail >>

Expected behavior might be that a command call would only run once.

<< creation behavior of commands >>

If the parent command runs it might automatically executes the calls inside it.

<< detail >>

When a call might be placed directly inside an *object,* it mightbe a question what would actually happen to it. It may be an idea, that the command might only be run manually. Another idea might be, that those calls might run just after the object was created (perhaps a bit like constructors from object oriented programming).

<< commands compared to objects >>

Because the definition is the prototype of the call, and not the same object as the call, a dashed class line needs to be used to point out the definition of a call.

<< already covered >>

A call does not have to be placed inside another command. It can also be placed inside an object, in case of which somebody has to run the executable object manually.

### Command Reference Behavior

<< commands compared to objects >>

A command reference might be achieved with *object* redirection.

<< detail >>

A command reference may commonly be inactive. But a command reference might also be active.

<< explains other technology >>

The handy thing about command reference, is that it makes you able to keep the operation to execute variable. The target of the command reference is variable. When you *call* a command reference, then the target of the command reference determines which command is called. So calling a command reference means calling a variable command definition.

<< commands compared to objects >>

A command reference represents the same object as the object pointed to. So the command reference’s contents are the exact same objects as that of the command object it points to.

<< detail >>

Both the command reference and its target can be either active or inactive.

<< detail >>

An *active* reference to an *inactive* command might not be executed. The final target of object redirections is the object itself, and when it is inactive, the command object can not be executed, but only function as a prototype.

You can not execute an *active* command object through an *inactive* command reference either. But an active reference to an inactive reference to an active command *can* be executed.

### Command Reference Pointer to Pointer Situations

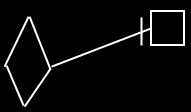
<< move >>

A command reference can also redirect to yet another command reference, creating multiple command object redirections. The target of the last command reference determines the definition of the first command reference.

### Command Reference Edge Cases

<< detail >>

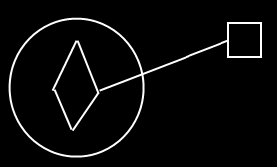
In Circle command references and their target commands could in theory be either inactive or executable.



It might not matter whether it is squares or diamonds, because the only difference between a square and a diamond, is that a square can notbe executed and a diamond *can*.

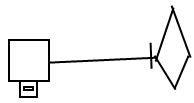
<< already covered >>

Direction of the line must be indicated with an access mark, unless the line is going outwards:



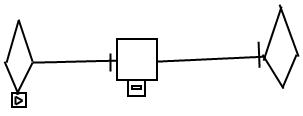
<< detail >>

You can not execute an active command object through an inactive command reference.



<< detail >>

But with an executable reference to an inactive reference to an executable command you *can* execute the command object again.



### Clause

#### Concept

<< already covered >>

A clause is a command defined within a command. A clause does not redirect its definition or object. A clause can also reside inside *yet* *another* clause.

<< detail >>

A clause has a fixed logical residence inside another command.

<< already covered >>

A clause can be *active* or *inactive*. If it is active, it is like a command call, executed when its parent command is executed. If a clause is *inactive*, then it is only executed when it is *called*.

<< creation behavior of commands >>

A clause is always created as long as the parent command is created.

<< detail >>

A clause does not redirect its definition, because then it would be a command call.

<< detail >>

A clause does not redirect its object, because then it would be a command reference.

<< detail >>

A clause is never situated inside an object, or it would not be a clause.

<< detail >>

Clauses are like command definitions, therefore they can have parameters just like command definitions.

#### Diagram Notation

<< already covered >>

A clause is a command, defined within a command.

<< already covered >>

An active clause is shown in a diagram as a diamond placed inside a command symbol:



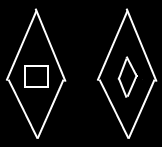
<< already covered >>

An *inactive* clause is shown in a diagram as a square inside a command symbol.



<< already covered >>

A clause can just as well be placed inside a diamond, instead of a square:

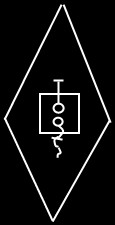


<< already covered >>

A clause will never redirect its definition or object to another command.

<< broader perspective >>

Clauses can have parameters, just like command definitions:



### Inactive Clause

<< already covered >>

A clause is a command defined within a command.

<< detail >>

An *inactive* clause is only executed when it is called. An inactive clause can be called and referenced like any other command.

<< already covered >>

It is common to pass a reference to an inactive clause to an execution control command, such as a conditional execution or a loop.

<< detail >>

An inactive clause *is* a command definition, with a fixed logical residence inside another command.

#### In a Diagram

<< detail >>

A clause is a command defined within a command.

<< already covered >>

An inactive clause is shown in a diagram as a square inside another command symbol.



<< already covered >>

A clause can just as well be placed inside a diamond, instead of a square:



<< already covered >>

A clause will never redirect its definition or object to another command.

### Active Clause

#### Concept

<< already covered >>

A clause is a command defined within a command.

<< already covered >>

An *active* clause executes when its parent command is executed.

<< details >>

An active clause is actually more primitive than a command call, but command call was explained first, because it is more commonly used.

<< detail >>

An active clause is like a call and a definition at the same time.

<< commands compared to objects >>

An active clause is analogous to an object that does not have a class. A *call* is more like an object that *does* have a class.

<< nice formulation >>

Because an [active clause] has its own definition, its contents are totally arbitrary and definable by the author of the parent command, unlike calls, whose contents comply with the definition, that they call.

<< detail >>

An active clause can not be redirected, because that would turn it into a command reference and not make it be an active clause anymore.

<< repeated >>

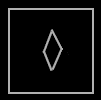
Active clauses can freely be used, to put a frame around a piece of code inside a command. In that case, the whole command will still do exactly the same thing.

#### Diagram Notation

<< already covered >>

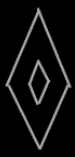
A clause is a command defined within a command.

An active clause is shown in a diagram as a diamond placed inside a command symbol:



<< already covered >>

A clause can just as well be placed inside another diamond, instead of a square:



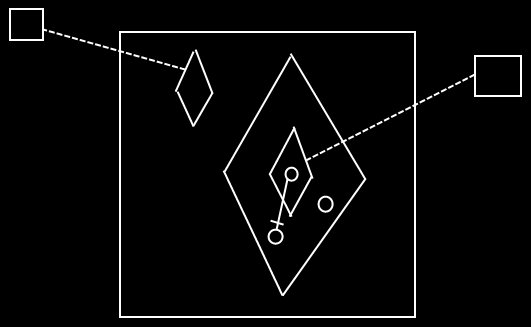
<< detail >>

A clause will never redirect its definition or object to another command.

<< nice formulation >>

Active clauses can freely be used to put a frame around a piece of code inside a command. In that case the whole command will still do exactly the same thing:





### Commands Compared to Objects

<< commands compared to objects >>

#### Commands Anywhere

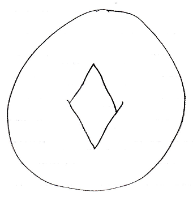
Commands are executable objects that can freely move around. You have to start thinking of an executable object more like an object, that just happens to be executable. You are going to have to see an execution much, much more like an executable object that can be situated anywhere and referenced from anywhere. You can put the executable object inside another object. You can reference an executable object. Yes, you can put an executable object inside a command definition. But you can also put an executable object inside a class.

You can run an execution inside an object, sort of like having an execution run inside a specific folder. You can also run the execution inside your 'user object' and sort of take it along with you, wherever you go into the digital world. You can also run an execution globally on a site.

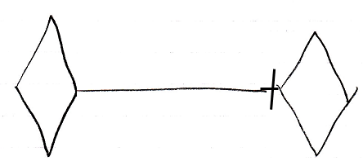
Executions can also be placed in a list. No problem. You can run each item in the list individually.

##### Diagram Notation

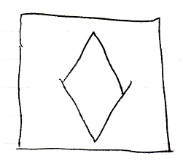
You can put the executable object inside another object:



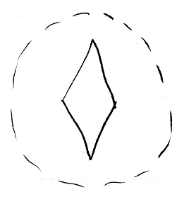
You can reference an executable object:



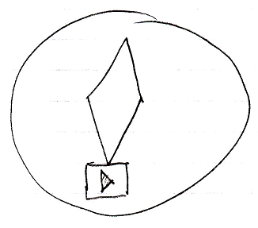
Yes, you can put an executable object inside a command definition:



But you can also put an executable object inside a class:

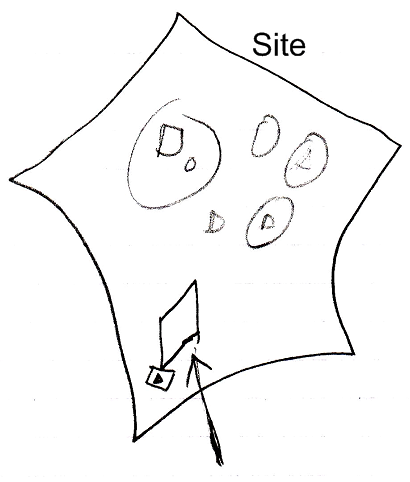


You can run an execution inside an object, sort of like having an execution run inside a specific folder:

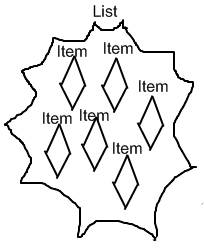


You can also run the execution inside your user object and sort of take it along with you, where ever you go into the digital world.

You can also run an execution globally on a site.



Executions can also be placed in a list. No problem. You can run each item in the list individually.



#### Resolution When Not Allowed For Commands

When the command concept imposes a rule, that does not apply to normal objects, then you will seem to be able to break that rule. To keep direct conversion between an object and a command possible without any loss of structure, it seems to be allowed to break any of the extra rules imposed by commands. But in that case the command will always get the next best alternative behavior and a warning will be generated. This keeps conversion between objects and commands possible without any loss of structure.

### Commands Edge Cases

#### Changing Inactive to Executable

<< repeated >>

Any part of a system can be changed, so an inactive command can be edited and changed to an executable command and back again.

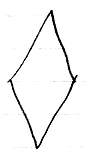
When you design a command definition, you might want to do it using an active command object, and test it once by running the active command definition. After that you can change it to an *inactive* command definition and give it the appropriate default values again. If you decide to again change the inactive command definition *back* to an active command, then you can not run it again, because the command object has already been run. Others might still have a reference to the execution to be able to read its output. If you want to run the command definition again, you are going to have to make a call to it instead.

If you change an inactive command definition to an executable command definition, then you can all of a sudden run it, which may overwrite the executable’s default output values.

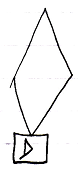
##### Diagram Notation

<< details >>

When you design a command definition, you might want to do it using an active command object:



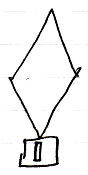
and test it once by running the active command definition:



After that you can change it to an inactive command definition and give it the appropriate default values.

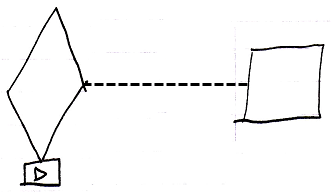


If you decide to change the inactive command definition back to an active command, then you can not run it again, because the command object has already run:

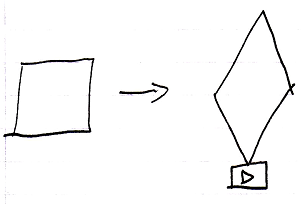


Others might still have a reference to it to be able to read its output.

If you want to run the command definition again, you are going to have to make a call to it instead:



If you change an inactive command definition to an executable command definition:



then you can all of a sudden run it, which may overwrite the executable’s default output values.

### Sub-Commands

<< details >>

Executions of commands can run inside a parent command. Sub-commands are *active* commands contained inside a *parent* command.

<< details >>

<< broader perspective >>

<< already covered >>

(in chapter "Introduction" and article "Creation Behavior of Commands")

<< move >>

(to article "Creation Behavior of Commands")

A parent command automatically executes its sub-commands. After a sub-command completes, the process returns to the parent command, which will then continue, executing the next sub-command.

Inside a command, usually just more commands are invoked.

There are only a few commands that do something other than execute other commands. Those are special commands, that perform a machine instruction: an operation that is executed by the CPU, the central processing unit of the computer.

On top of those special commands, a few basic commands exist, like **If**’s and **For** loops, that control the flow of a program, making the next command to call dependent on a condition.

But basically, a command just calls more commands. Machine instructions, arithmetic operators, comparative and Boolean algebra, assignments, and execution control statements such as **If** and **For**, are *all just commands*.

<< broader perspective >>

Apart from sub-commands, a command can also contain data.  
(Inactive clauses and inactive command references for instance are also considered data, and are not sub-commands, because they do not execute.)

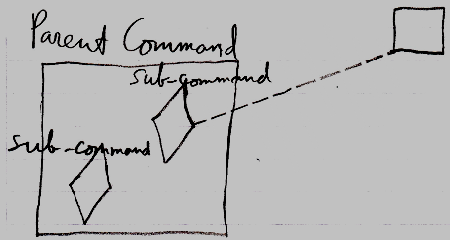
#### Sub-Commands in a Diagram

<< details >>

(terminology quibbles)

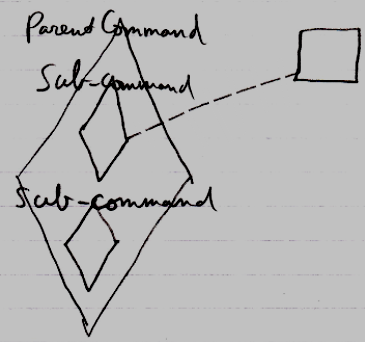
Sub-commands are active commands contained inside a parent command. You can encounter them inside any command symbol: both active and inactive commands symbols can contain sub-commands. The sub-commands are *active* commands: command calls, active clauses or active command references.

Below is a picture of two sub-commands inside an inactive command:



The **Parent Command** is an inactive command, because it is a square. In the diagram above, the **Parent Command** contains two sub-commands. The sub-commands are displayed as diamond shapes. One of the sub-commands is a call, because it has a dashed line going outside the **Parent Command**, tying the sub-command to its command definition. The other sub-command is an *active clause*, because it does not redirect its command definition.

Below is a picture of two sub-commands inside an active command:



It is the same picture as the other diagram, only now the **Parent Command** is an executable command symbol, not an inactive command.

### Command References Inside Commands

<< details >>

A command reference is considered data. A command reference in a parent command is considered data of the parent command, even when it is active, in case of which it *is* considered a sub-command but a special form of it. Even though it executes, it is still a reference to an executable object *elsewhere*. Only the *pointer* part is the data. Command references are *data* that can be *private* or *public*.

### Executables & Executions

<< repeated >>

An executable object stands for a potential execution.

<< repeated >>

An executable object never stands for multiple executions, just one execution or no execution at all.

<< repeated >>

So each execution is represented by its own individual executable object.

<< details >>

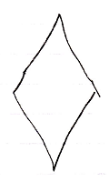
Only when an executable object is actually executed, it is called an execution. When an executable object is not executing yet, it is still only a *potential* execution.

#### Diagram Notation

<< details >>

(terminology quibbles)

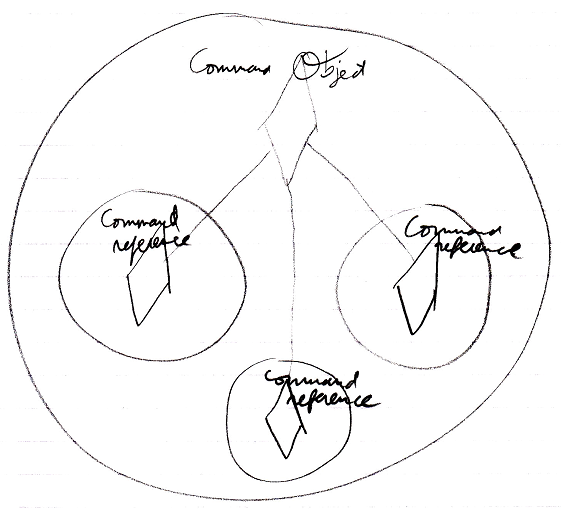
An executable object stands for a potential execution:



An executable object never stands for multiple executions, just one execution or no execution at all.

<< commands compared to objects >>

The same command object can be displayed in the diagram multiple times. In that case, a symbol might be added to the diagram, that the multiple references to the same command object will converge to, so that a single symbol in the diagram is selected to represent the command object itself, while the other ones are just references.



So it is not so, that each diamond in the diagram represents its own individual command execution. Each individual *command object* represents an individual command execution. Multiple symbols in the diagram can represent the same command object, but will converge into one symbol representing the object itself.

### Public Inactive Clause = Command Out Parameter

<< repeated >>

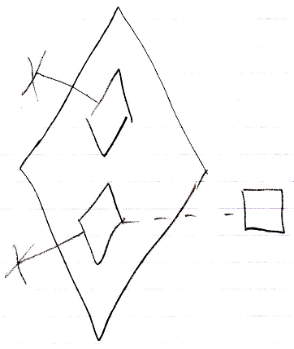
*Active* clauses, command calls and active command references in parent commands are always private, because you can not reference a sub-command.

But *inactive* clauses *can* be referenced and might be made public. If you make an inactive clause public, you will make it an **Object Out** parameter: an object produced or determined by the command. A public inactive clause would be a command definition produced by another command.

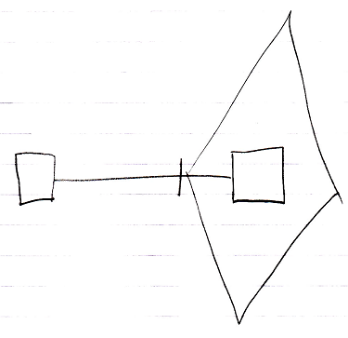
#### Diagram Notation

<< detail >>

Active clauses, command calls and active command references in parent commands are always private, because you can not reference a sub-command.



But *inactive* clauses *can* be referenced and might be made public.



If you make an inactive clause public, you will make it an **Object Out** parameter: an object produced or determined by the command. It would be an **Object Out** parameter, that is an executable object. That would work just fine.

### Reading & Writing Parameters

<< parameters >>

Whether a parameter of a command can be read or written at all is access-controlled. This access control determines whether a parameter is for instance input or output.

<< creation behavior of commands >>

Before a command is run you can mess about with the parameters all you want.

During the execution of a command you can not read or write anything.

After a command has executed, you can not change the parameters, just out of practical reasons, because it is more practical for the parameters to keep visualizing the state they were in after the command was run.

- Before execution you can read and write.

- During execution you can not read or write.

- After execution you can only read.

<< commands compared to objects >>

A command definition’s parameter values are public, so if you can reach the definition, you can basically change its parameter values, and mess about with them all you want. This needs to be access controlled, but how this should be done is not yet determined.

<< parameters >>

Details about parameters can be found in the *Parameters* articles.

### Exchangeability Between Commands & Objects

<< commands compared to objects >>

In some programming languages, commands might not be exchangeable with objects, but be totally separate constructs. But in the Circle, commands could be seen as objects, that just happen to be executable.

Using some programming languages a process might be implemented as a procedure. Sometimes a procedure might become more complex. It might be an option to then rewrite a procedure to an object, whose purpose could be to execute that procedure. This object may then hide complexity of the input and output better, and might better hides the complexity of the procedure itself. An object in some programming languages might offer a way to split up a procedure into separate steps, without seeing that on the outside. Turning a procedure into an object may make linking to it more flexible. For instance you might plug a new process into an already existing system, that was not aware of this procedure. That might not be easily possible, unless you turn a procedure into an object.

In Circle though, procedures might be considered an object as is, without rewriting anything. The only difference might be that this object has the special property, that it would be *executable*.

In the Circle notation, a command could have complexity hiding and linking possibilities, just like a 'normal' object. A command might for instance also *inherit* a base procedure from another command, just to name something.

A command could be structured similarly as an object. Perhaps that takes away the argument that using just procedures might be a less structured way of programming than using objects. A command might already be an object, that represents a process. It may looks like a command, but might have the capabilities of an object.

In that philosophy, commands would be *executable objects*.

### Execute Once

<< creation behavior of commands >>

<< details >>

a command object might only execute *once*. A command object may represent a single execution. An execution might be created and dormant until it would be run. This might be a chance to set the input of the command. After a command was run, the execution might stay created while it may still be referenced, so output might be read, until everybody might be done with it. When the executable object would not be referenced anymore, it might be destroyed. A command object might not be executed twice: to run a command again, a new command object might be created, that may have the same *definition*. In an attempt to execute the same command symbol twice, what might happen instead, is that the old object may be released, and a new object could be created in its place. A reason why a command object might only be executed once, may be that this would give all the referrers a chance to read the execution’s output, whenever they want, without it being overwritten by new output. A command object stays created for as long as it is being referenced, so everybody can read the output of the command. The command object will only be destroyed when nothing refers to it anymore.

### Command Definition Analogy to Classes

<< commands compared to objects >>

Commands with the same definition contain the same list of attributes, related items and related lists.

There is a complete analogy between a command’s definition and the class of an object.

Anything that applies to classes, also applies to command definitions.

### No Dashed Squares?

<< basic diagram elements >>

When an *object* is drawn with a dashed line, then it is only used as a class:



This notation will *not* be copied to the concept of commands. This is because a command’s definition will usually be a square and a square will usually be a command’s definition, so using dashed lines for command definitions, would create an overload of dashed squares. So command symbols, that are only used as a definition, do not get a dashed notation.

<< details >>

A command, that might be used as a definition, is usually not executable. Only individual calls to the command, that use the command as a definition, might be executable. If a command definition is executable after all, then it is clearly mentioned, that it is an *active* command definition, because it might be a special situation.